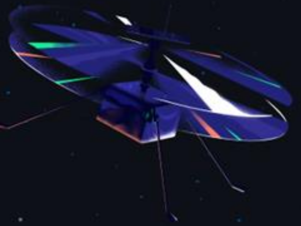
GitHub Copilot Training for Documentation



Andrew Scoppa

GitHub Copilot

Resources

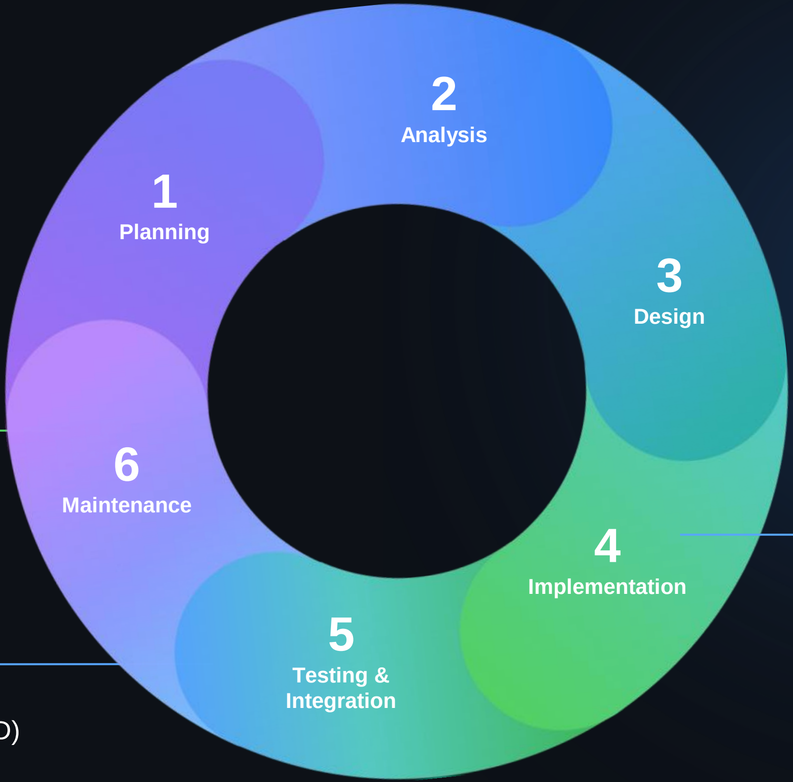
- Getting started with GitHub Copilot
- Configuring GitHub Copilot in your environment
- Using GitHub Copilot Chat in your IDE - GitHub Docs
- Tutorial: GitHub Copilot and VS Code

AGENDA

- GitHub Copilot - Introduction
- Prompt crafting
- In-class Demos
- Security
- Wrap-up, Q&A

GitHub Copilot Fundamentals Recap

Copilot + Copilot Chat



Refactoring code (code translate)
Reviewing code (code explain)
Documentation

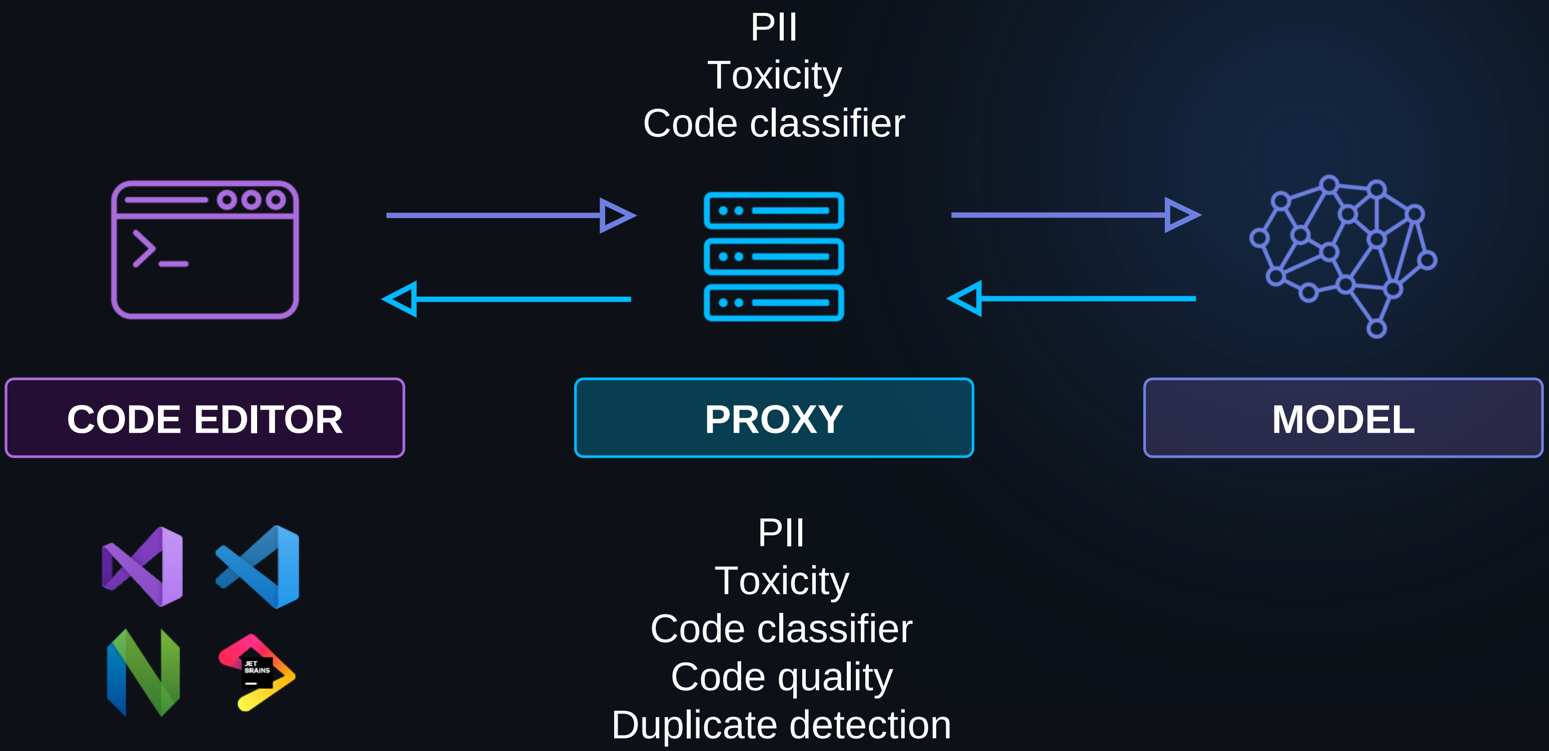
Convert comments to code
Autofill for repetitive code
Show alternatives

Unit testing (TDD and BDD)
Finding code errors
Debugging
Code review
AI Pull Requests

GitHub Copilot

- Like a smart assistant or mentor by your side
- Draws context from comments & code in open tabs
- Think of something like Google Translate
- Powered by OpenAI Codex
  - Copilot uses a transformative model
  - Trained on large datasets to ensure accuracy
  - **It even can help with HTML and markdown!**
- Available as an extension to IDEs and editors like VS Code

Data flow through the Copilot ecosystem



PII
Toxicity
Code classifier

CODE EDITOR

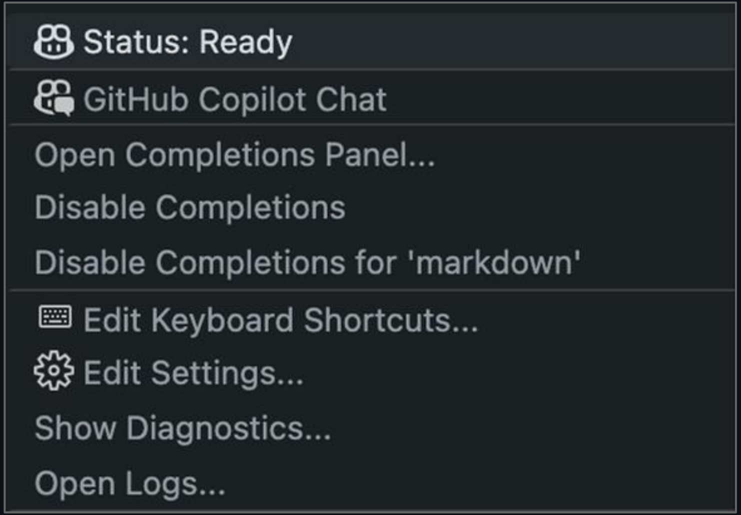
PROXY

MODEL

PII
Toxicity
Code classifier
Code quality
Duplicate detection

VS Code Settings

Configuration Options



[Configuring GitHub Copilot in your environment]

Prompt Engineering

What is Prompt Engineering?

Prompt engineering is the process of designing and creating high-quality prompts that can be used to generate accurate and useful suggestions with Copilot.

Providing Context

To help Copilot generate accurate suggestions:

- Add a top-level comment block describing the purpose of the file
- Write clear instructions
- Have related content open in other tabs

Prompting Techniques

In the realm of GitHub Copilot, Zero-shot, One-shot, and Few-shot prompting refer to guiding the AI with varying levels of examples.

Zero-shot prompting doesn't provide any prior examples, expecting Copilot to understand and generate relevant code purely from the given task description.

One-shot prompting provides a single example to set the context, assisting Copilot in generating a similar outcome.

Few-shot prompting involves offering multiple examples to establish a clearer pattern for the desired code output.

By understanding these techniques, developers can better instruct Copilot, ensuring more accurate and context-aware code suggestions.

Using Copilot Chat

GitHub Copilot Chat

- ○ Refactor Code
- ○ Generate Tests
- ○ Debug
- ○ Create Workspace
- ○ Documentation

Copilot vs Copilot Chat

Copilot

Direct Code Writing

Seamless IDE Integration

Solo Development

Copilot Chat

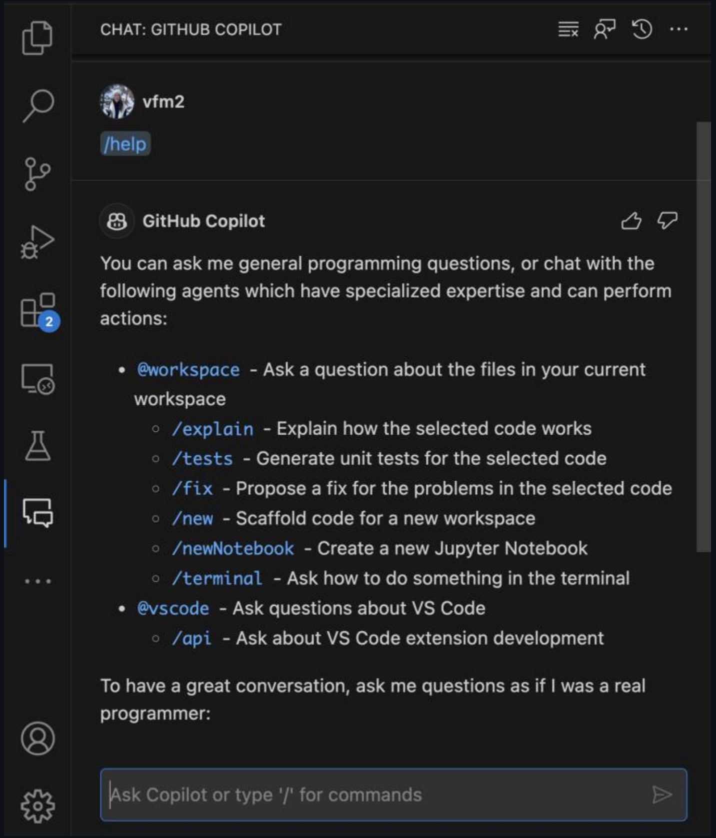
In-Depth Assistance

Learning & Teaching

Collaborative Scenarios

Copilot Chat: Slash Commands

/help to find available commands in your IDE



vfm2

/help

GitHub Copilot

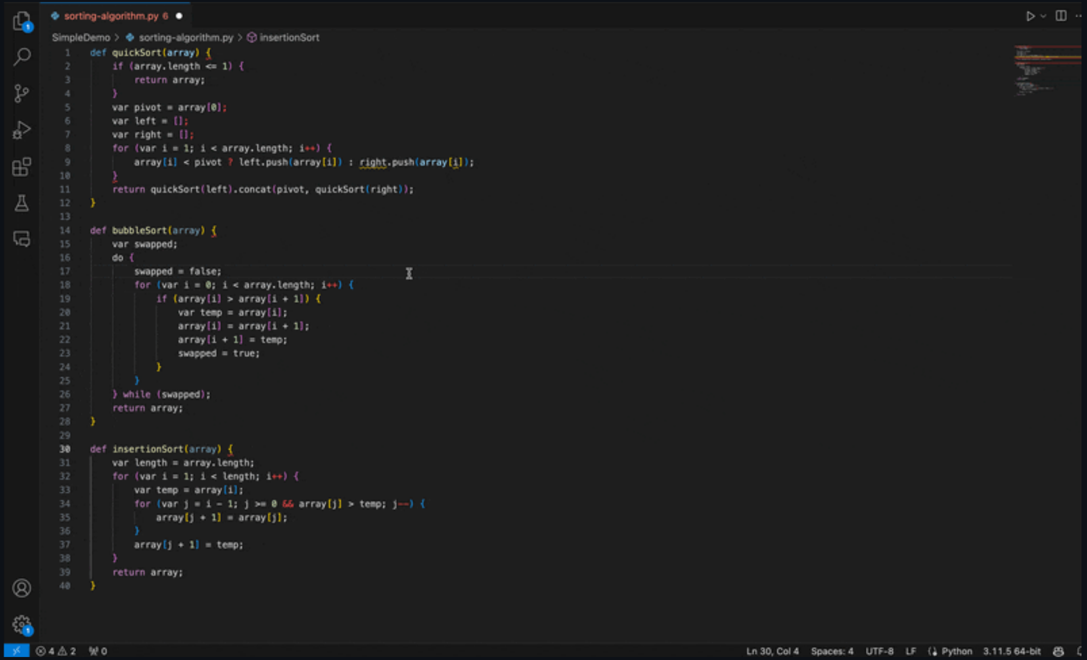
You can ask me general programming questions, or chat with the following agents which have specialized expertise and can perform actions:

- `@workspace` - Ask a question about the files in your current workspace
  - `/explain` - Explain how the selected code works
  - `/tests` - Generate unit tests for the selected code
  - `/fix` - Propose a fix for the problems in the selected code
  - `/new` - Scaffold code for a new workspace
  - `/newNotebook` - Create a new Jupyter Notebook
  - `/terminal` - Ask how to do something in the terminal
- `@vscode` - Ask questions about VS Code
  - `/api` - Ask about VS Code extension development

To have a great conversation, ask me questions as if I was a real programmer:

In-file Copilot Chat

Copilot offers in-file Copilot feature to selectively improve



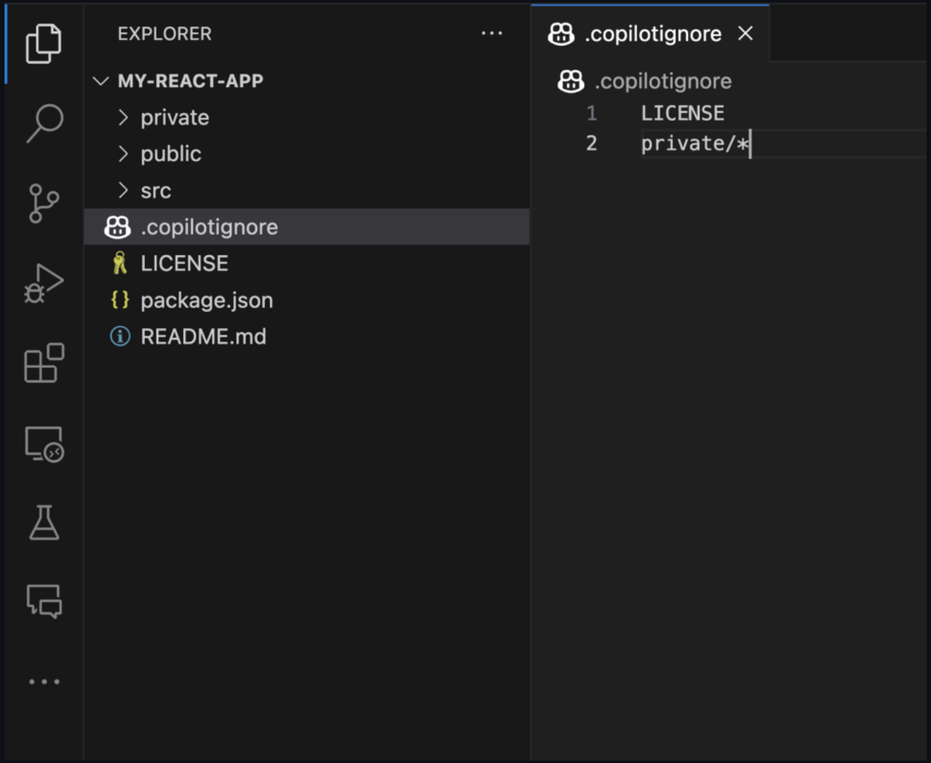
```
def quickSort(array) {
    if (array.length <= 1) {
        return array;
    }
    var pivot = array[0];
    var left = [];
    var right = [];
    for (var i = 1; i < array.length; i++) {
        array[i] < pivot ? left.push(array[i]) : right.push(array[i]);
    }
    return quickSort(left).concat(pivot, quickSort(right));
}

def bubbleSort(array) {
    var swapped;
    do {
        swapped = false;
        for (var i = 0; i < array.length; i++) {
            if (array[i] > array[i + 1]) {
                var temp = array[i];
                array[i] = array[i + 1];
                array[i + 1] = temp;
                swapped = true;
            }
        }
    } while (swapped);
    return array;
}

def insertionSort(array) {
    var length = array.length;
    for (var i = 1; i < length; i++) {
        var temp = array[i];
        for (var j = i - 1; j >= 0 && array[j] > temp; j--) {
            array[j + 1] = array[j];
        }
        array[j + 1] = temp;
    }
    return array;
}
```

GitHub Copilot Chat (default)

Security

Block files from Copilot

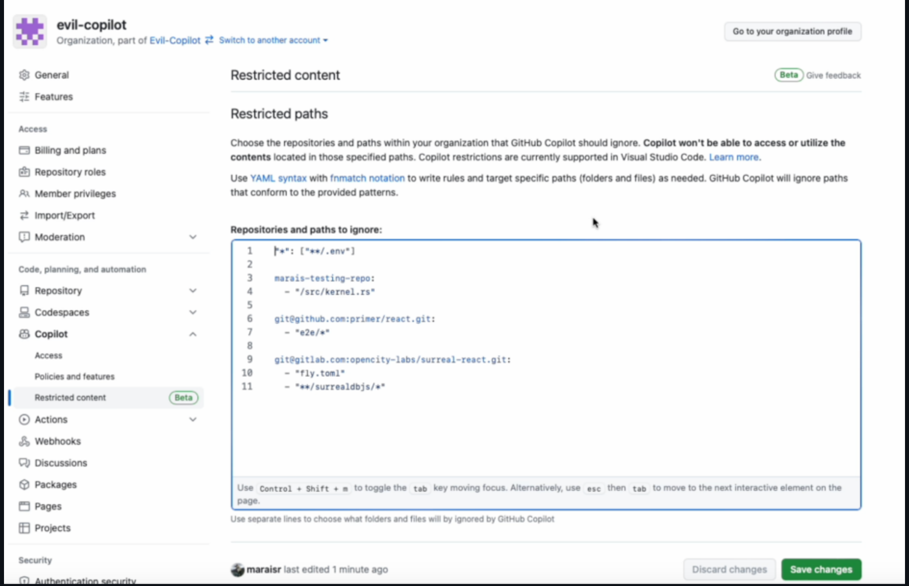
Use .copilotignore to block files and folders from being used by Github Copilot



```
LICENSE
private/*
```

Block files from Copilot

Ignore specific repositories and paths in Restricted Content



Restricted content

Restricted paths

Choose the repositories and paths within your organization that GitHub Copilot should ignore. **Copilot won't be able to access or utilize the contents** located in those specified paths. Copilot restrictions are currently supported in Visual Studio Code. Learn more.

Use YAML syntax with fnmatch notation to write rules and target specific paths (folders and files) as needed. GitHub Copilot will ignore paths that conform to the provided patterns.

Repositories and paths to ignore:

```
"*": ["**/.env"]

marais-testing-repo:
  - "/src/kernel.rs"

git@github.com:primer/react.git:
  - "e2e/*"

git@gitlab.com:opencity-labs/surreal-react.git:
  - "fly.toml"
  - "**/surrealdbjs/*"
```

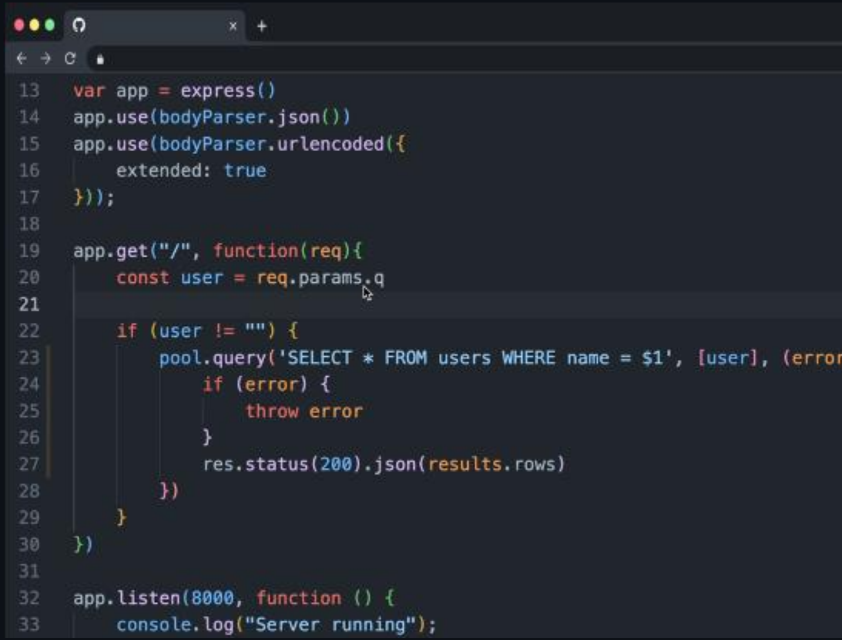
Use `Control + Shift + m` to toggle the `tab` key moving focus. Alternatively, use `esc` then `tab` to move to the next interactive element on the page.

Use separate lines to choose what folders and files will by ignored by GitHub Copilot

maraisr last edited 1 minute ago

Copilot and Secure Coding

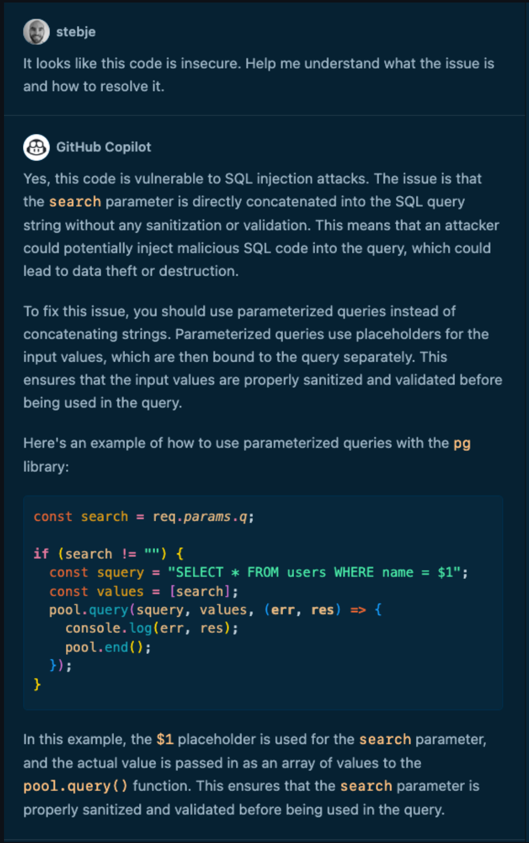
- AI-based vulnerability system that helps prevent insecure coding patterns (e.g. SQL script injection)
- Vulnerability filters are applied only to the Copilot-generated suggestion itself
  - ⊗ It cannot detect downstream vulnerabilities introduced by the code e.g. on deployment infrastructure
  - ⓘ **We recommend taking the same precautions you take with code written by engineers (linting, code scanning, etc.)**
- Copilot Chat can be used to query code for known vulnerabilities



```
var app = express()
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({
    extended: true
}));

app.get("/", function(req){
    const user = req.params.q

    if (user != "") {
        pool.query('SELECT * FROM users WHERE name = $1', [user], (erro
            if (error) {
                throw error
            }
            res.status(200).json(results.rows)
        })
    }
})

app.listen(8000, function () {
    console.log("Server running");
```

Detection & Remediation

- Prompt for insecure coding patterns
- Use GHAS Code Scanning results
- Write custom CodeQL queries
- Increase your knowledge of secure coding patterns
- Create custom Secret Scanning patterns



stebje

It looks like this code is insecure. Help me understand what the issue is and how to resolve it.

GitHub Copilot

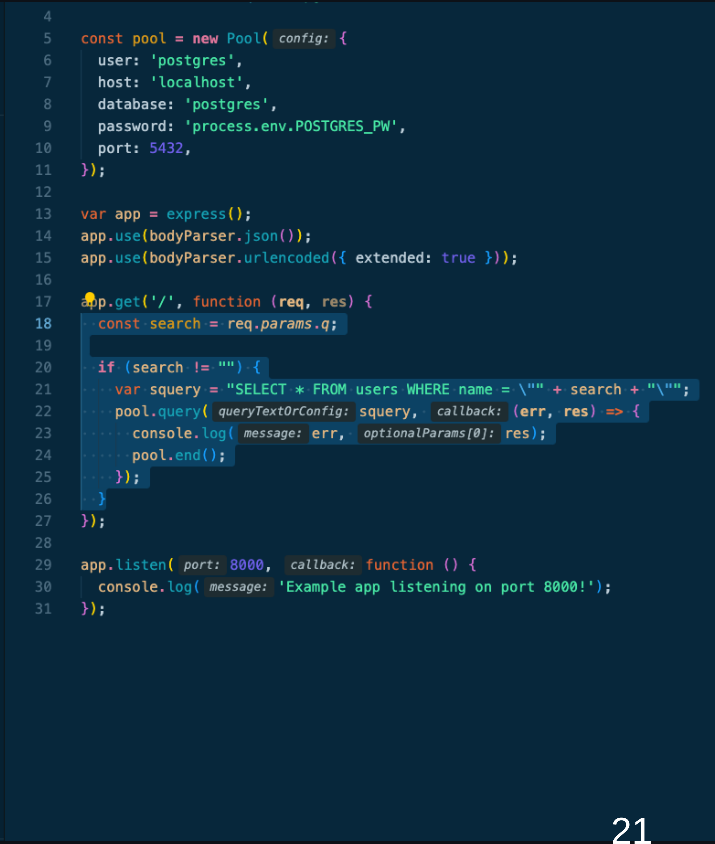
Yes, this code is vulnerable to SQL injection attacks. The issue is that the `search` parameter is directly concatenated into the SQL query string without any sanitization or validation. This means that an attacker could potentially inject malicious SQL code into the query, which could lead to data theft or destruction.

To fix this issue, you should use parameterized queries instead of concatenating strings. Parameterized queries use placeholders for the input values, which are then bound to the query separately. This ensures that the input values are properly sanitized and validated before being used in the query.

Here's an example of how to use parameterized queries with the `pg` library:

```
const search = req.params.q;

if (search != "") {
  const squery = "SELECT * FROM users WHERE name = $1";
  const values = [search];
  pool.query(squery, values, (err, res) => {
    console.log(err, res);
    pool.end();
  });
}
```

In this example, the `$1` placeholder is used for the `search` parameter, and the actual value is passed in as an array of values to the `pool.query()` function. This ensures that the `search` parameter is properly sanitized and validated before being used in the query.

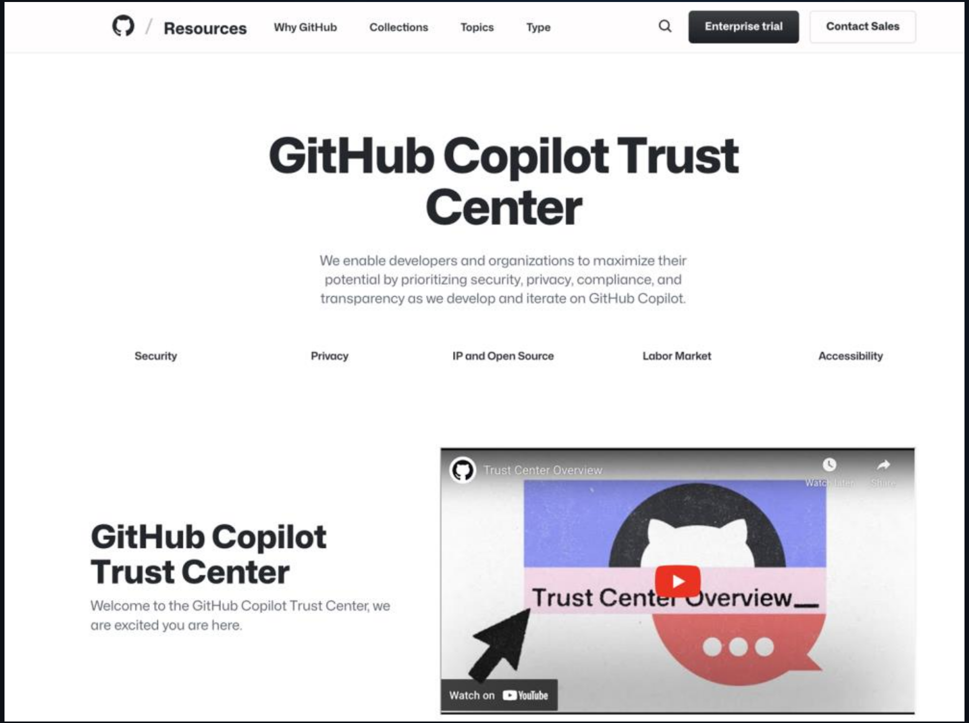


```

const pool = new Pool( config: {
  user: 'postgres',
  host: 'localhost',
  database: 'postgres',
  password: 'process.env.POSTGRES_PW',
  port: 5432,
});

var app = express();
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.get('/', function (req, res) {
  const search = req.params.q;

  if (search != "") {
    var squery = "SELECT * FROM users WHERE name = \"" + search + "\"";
    pool.query( queryTextOrConfig: squery,  callback: (err, res) => {
      console.log( message: err,  optionalParams[0]: res);
      pool.end();
    });
  }
});

app.listen( port: 8000,  callback: function () {
  console.log( message: 'Example app listening on port 8000!');
});
```

Security & Trust

Copilot Trust Center

- Security
- Privacy
- Data flow
- Copyright
- Labor market
- Accessibility
- Contracting



GitHub Copilot Trust Center

We enable developers and organizations to maximize their potential by prioritizing security, privacy, compliance, and transparency as we develop and iterate on GitHub Copilot.

GitHub Copilot Trust Center

Welcome to the GitHub Copilot Trust Center, we are excited you are here.

GitHub Copilot Trust Center - GitHub Resources

Wrap Up

Thank you